2INC0 - Operating Systems

Actions synchronization

Geoffrey Nelissen

IRiS
Interconnected
Resource-aware
Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Announcements

- Second homework is available (deadline on Sunday)
- If you have questions about the practical assignments, contact Olav Bunte

A note on “zombies”

© PopCap studio

- A zombie is a child process that terminated, but the parent does not call wait() or wait_pid()
- It does not matter what the child does (i.e., whether it calls exec() or any other system call or function)

```
int main() {

    int i = 0;

    for (int k=0; k< 5;  ++k) {
        fork();
        execlp("/bin/sh","/bin/sh", "-c", "ls -l /bin/??", (char *)NULL);
    }

    i++;
    return 0;
}
```

Course Overview

- What is an OS
- Purpose of an OS

- How to implement concurrency (for efficiency)

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduling** (lectures 2+3)
- **Concurrency and synchronization**
    - atomicity and interference (lecture 4)
    - **actions synchronization (lecture 5)**
    - condition synchronization (lecture 6)
    - deadlock (lecture 7)

- Dangers of concurrency (**race conditions**)
- How to prove properties using **traces**
- Synchronization (**critical sections**)

- **File systems** (lecture 8)
- **Memory management** (lectures 9+10)
- **Input/output** (lecture 11)

Agenda

How can we **check** complex **properties** **without checking all possible traces** of a concurrent program

- **Proving program properties using invariants**
- **Actions synchronization using semaphores**
- **Preventing deadlocks**

How can we **enforce properties** during the program execution **using semaphores**

Topology invariants

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program** code
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

```
int x = 1;
int y = 0;
```

```
Task1 = |[                      ||    Task2 = |[
    while (true) {                        while (true) {
        < x++; >                              < y--; >
        < y++; >                              < x--; >
    }                                     }
]|                                    ]|
```

Topology invariants

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program code**
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

```
int x = 1;
int y = 0;
```

$Task_1$ = |[
  while (*true*) {
    X_1: < x++; >
    Y_1: < y++; >
  }
]|

||

$Task_2$ = |[
  while (*true*) {
    Y_2: < y--; >
    X_2: < x--; >
  }
]|

cX = number of times action *X* executed

I1: $0 \leq cX_1 - cY_1$

Number of times Y_1 executes is **never more than** the **number of times X_1** executes

Topology invariants

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program code**
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

int *x = 1;*
int *y = 0;*

Task$_1$ = |[
    while (*true*) {
        X_1: < x++; >
        Y_1: < y++; >
    }
]|

||

Task$_2$ = |[
    while (*true*) {
        Y_2: < y--; >
        X_2: < x--; >
    }
]|

cX = number of times action *X* executed

I1: $0 \leq cX_1 - cY_1 \leq \mathbf{1}$

I2: $0 \leq cY_2 - cX_2 \leq 1$

Number of times Y_1 executes is **never more than** the **number of times X_1** executes

Number of times Y_1 executes is **at most one time less than X_1**

Topology invariants

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program** code
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

```
int x = 1;
int y = 0;
```

```
Task1 = |[
    while (true) {
        X1: < x++; >
        Y1: < y++; >
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: < y--; >
        X2: < x--; >
    }
]|
```

cX = number of times action *X* executed

I1: $0 \leq cX_1 - cY_1 \leq 1$

I2: $0 \leq cY_2 - cX_2 \leq 1$

I3: $y = ?$

Topology invariants

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program code**
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

```
int x = 1;
int y = 0;
```

```
Task1 = |[
    while (true) {
        X1: < x++; >
        Y1: < y++; >
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: < y--; >
        X2: < x--; >
    }
]|
```

cX = number of times action *X* executed

I1: $0 \leq cX_1 - cY_1 \leq 1$

I2: $0 \leq cY_2 - cX_2 \leq 1$

I3: $y = cY_1 - cY_2$

I4: $x = ?$

Topology invariants

TU/e

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program code**
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

```
int x = 1;
int y = 0;
```

```
Task1 = |[
    while (true) {
        X1: < x++; >
        Y1: < y++; >
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: < y--; >
        X2: < x--; >
    }
]|
```

cX = number of times action *X* executed

I1: $0 \leq cX_1 - cY_1 \leq 1$

I2: $0 \leq cY_2 - cX_2 \leq 1$

I3: $y = cY_1 - cY_2$

I4: $x = 1 + cX_1 - cX_2$

IRiS

Using topology invariants to prove program properties

cX = number of times action X executed

```
int x = 1;
int y = 0;
```

```
Task1 = |[
    while (true) {
        X1: < x++; >
        Y1: < y++; >
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: < y--; >
        X2: < x--; >
    }
]|
```

I_1: $0 \leq cX_1 - cY_1 \leq 1$

I_2: $0 \leq cY_2 - cX_2 \leq 1$

$\mathbf{I_3}$: $y = cY_1 - cY_2$

$\mathbf{I_4}$: $x = 1 + cX_1 - cX_2$

Prove that $x > y$

using I_3 and I_4

$\Leftrightarrow 1 + cX_1 - cX_2 > 0 + cY_1 - cY_2$

$\Leftrightarrow 1 + cX_1 + cY_2 > 0 + cY_1 + cX_2$

by $I_{1:}$ $cX_1 \geq cY_1$, by I2 $cY_2 \geq cX_2$, and $1 > 0$

$\Rightarrow$ *true*

Exercise

```
int x = 4;
int y = 0;
int z = 1;
```

```
Task1 = |[                    ||   Task2 = |[                    ||   Task3 = |[
    while (true) {                     while (true) {                     while (true) {
        Z1: < z := z+2; >                  Y2: < y := y+1; >                  X3: < x := x+3 >
        X1: < x := x+2; >                  Z2: < z := z-1; >                  Y3: < y := y+3 >
    }                                  }                                  }
]|                                 ]|                                 ]|
```

Prove that $x \geq y + z$

Solve the problem together with your neighbors.

- You have **10 minutes**.

Exercise

```
int x = 4;
int y = 0;
int z = 1;
```

```
Task1 = |[
   while (true) {
      Z1: < z := z+2; >
      X1: < x := x+2; >
   }
]|
```

||

```
Task2 = |[
   while (true) {
      Y2: < y := y+1; >
      Z2: < z := z-1; >
   }
]|
```

||

```
Task3 = |[
   while (true) {
      X3: < x := x+3; >
      Y3: < y := y+3; >
   }
]|
```

Prove that $x \geq y + z$

- x = 4+ 2 * cX1 + 3 * cX3
- y = cY2 + 3* cY3
- z = 1 + 2*cZ1 – cZ2

- I1 : cZ1 - cX1 ≤ 1
- I2 : cY2 - cZ2 ≤ 1
- I3: 0 ≤ cX3 - cY3

➔ Replacing x, y and z using the topology invariant above, we get that $x \geq y + z$ is equivalent to proving that 4 + 2 * cX1 + 3 * cX3 ≥ cY2 + 3 * cY3 + 1 + 2 * cZ1 – cZ2
➔ Rewriting: 3 + 3 (cX3 - cY3) ≥ (cY2 – cZ2) + 2 (cZ1 – cX1)
➔ Using I2, the above is true if: 3 + 3 (cX3 - cY3) ≥ 1 + 2 (cZ1 – cX1)
➔ Using I3, the above is true if: 3 ≥ 1 + 2 (cZ1 – cX1)
➔ Using I1, the above is true if: 3 ≥ 3
➔ true

Agenda

- **Proving program properties using invariants**
- **Actions synchronization using semaphores**
- **Preventing deadlocks**

Example of a synchronization problem


```
int BufferCnt := 0;
```

```
Task Producer =
|[ while( true ){
        produce_items();

        BufferCnt := BufferCnt+4;
        push_items;
   }
]|
```

||

```
Task Consumer =
|[ while( true ){

        BufferCnt := BufferCnt-2;
        pop_items();
        use_items();
   }
]|
```

What can go wrong with this program?

Example of a synchronization problem

***int** BufferCnt := 0;*

Task *Producer* =
|[**while**(*true*){
        produce_items()*;*

        BufferCnt := BufferCnt+4;
        push_items();
   }
]|

||

Task *Consumer* =
|[**while(** *true*){

        BufferCnt := BufferCnt-2;
        pop_items();

What is the **solution** for the first one?
Use a mutex to protect the **critical sections**

1. **Race condition** when *Producer* and *Consumer* update the value of BufferCnt and when they push and pop items in the buffer
2. *Consumer* may try to **read from an empty buffer**
3. *Producer* may try to **write in a buffer that is full** (results in lost items)

Example of a synchronization problem

What are the potential issues with this solution?

Busy-wait

```
mutex m := 1;
int BufferCnt := 0;
Task Producer =
|[ while( true ){
        produce_items();
        while(BufferCnt > BufferSize -4)
                skip;
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
}
]|
```

||

```
Task Consumer =
|[ while( true ){
        while(BufferCnt < 2)
                skip;
        lock(m);
        BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        use_items();
   }
]|
```

1. **Race condition** when *Producer* and *Consumer* update the value of BufferCnt
2. *Consumer* may try to **read from an empty buffer**
3. *Producer* may try to **write in a buffer that is full** (results in lost items)

When should we busy-wait?

Busy-waiting Repeated testing without going to sleep

- Acceptable only **when**
  - waiting is guaranteed to be **short** in comparison to the cost of context switching or
  - there is **nothing else to do** (e.g., when the task executes on dedicated hardware)

- Busy-waiting using *while(.) skip;* works **only if**
  - the tests in the while and locking the mutex are **executed** atomically

Why?

Alternative solution:
use semaphores for actions synchronization

Recall semaphores (Dijkstra)

- **Non-negative integer** *s* with initial value s_0 and *atomic* operations *P(s)* and *V(s).*

  P(s): < await(s>0); s := s-1 > → the task **sleeps until '*s>0*'** holds, decrement *s*
  V(s): < s := s+1 > → increment *s*

- From a theoretical perspective, semaphores are always positive

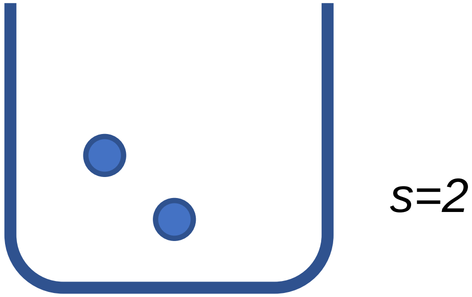
Note1: I use the notations *P(s)* and *V(s)*, and *lock(s)* and *unlock(s)* interchangeably when *s* is a **mutex** (i.e., a **binary** semaphore).

Note2: the reference book discusses the possibility to implement semaphores that become negative. The goal of such implementation is to record the number of tasks waiting on the semaphore (i.e., if *s* = *-5*, then five tasks are waiting on semaphore *s*). It does not have any theoretical implication and may not be supported in all OSs or programming languages. Therefore, we will always assume that a semaphore does not become negative in the lectures.

Semaphore (visual representation)

- Non-negative integer *s*
- Two atomic actions *P(s)* and *V(s)*

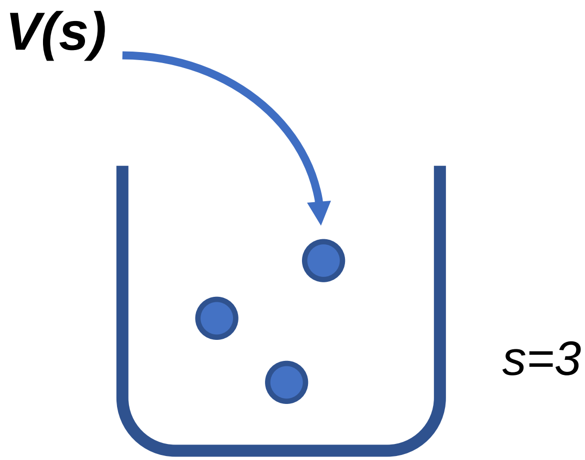
- A semaphore may be seen as a **bucket containing s tokens**



Semaphore (visual representation)

- Non-negative integer *s*
- Two atomic actions *P(s)* and *V(s)*

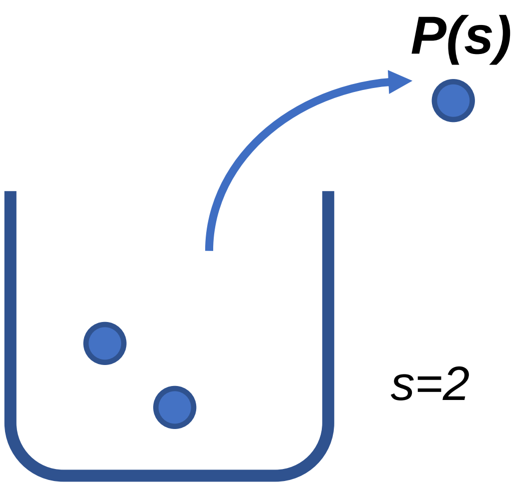
- A semaphore may be seen as a **bucket containing tokens**
- ***V(s) adds a token in the bucket***



Semaphore (visual representation)

- Non-negative integer *s*
- Two atomic actions *P(s)* and *V(s)*

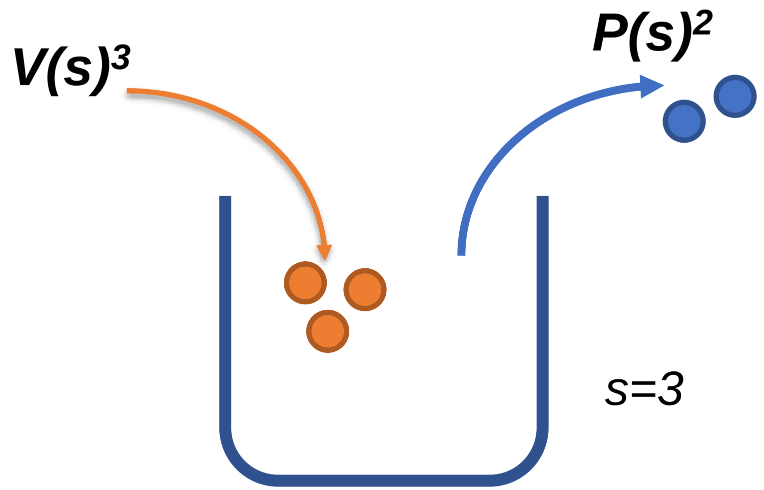
- A semaphore may be seen as a **bucket containing tokens**
- ***V(s) adds a token in the bucket***
- ***P(s) takes a token*** *from the bucket* ***if there is one in. Otherwise, it waits until one is added and takes it.***



Semaphore (visual representation)

- Non-negative integer *s*
- Two atomic actions *P(s)* and *V(s)*

- A semaphore may be seen as a **bucket containing tokens**
- ***V(s) adds a token in the bucket***
- ***P(s) takes a token*** *from the bucket* ***if there is one in. Otherwise, it waits until one is added and takes it.***
- ***V(s)***^{x} and ***P(s)***^{x} places or takes ***x*** tokens in or from the bucket



Action Synchronization Solution

Given:

- a collection of tasks executing a collection of actions A_i and B_j *(for i=0, 1, 2, … and j=0, 1, 2, …)*,
- and a required *synchronization condition (invariant)*
  Invariant: $\sum_i a_i \times cA_i \leq \sum_j b_j \times cB_j + e$
  for non-negative constants a_i, b_j, and e .

Solution:

- introduce semaphore *s*, with initial value $s_0 = e$
- replace action A_i with $P(s)^{a_i} A_i$
- replace action B_j with $B_j\, V(s)^{b_j}$

Works because of the safety property of semaphores (**see video**)

Example of a synchronization problem


```
mutex m := 1;
int BufferCnt := 0;
```

```
Task Producer =                         Task Consumer =
|[ while( true ){                   ||   |[ while( true ){
        produce_items();
                                                lock(m);
        lock(m);                                BufferCnt := BufferCnt-2;
        BufferCnt := BufferCnt+4;               pop_items();
        push_items();                           unlock(m);
        unlock(m);                              use_items();
}                                          }
]|                                      ]|
```

1. **Race condition** when *Producer* and *Consumer* update the value of BufferCnt and when they push and pop items in the buffer
2. *Consumer* may try to **read from an empty buffer**
3. *Producer* may try to **write in a buffer that is full** (results in lost items)

Example of a synchronization problem


```
mutex m := 1;
int BufferCnt := 0;
```

```
Task Producer =
|[ while( true ){
        produce_items();

        lock(m);
  G     BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
}
]|
```

||

```
Task Consumer =
|[ while( true ){

        lock(m);
  C     BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        use_items();
  }
]|
```

Equivalent synchronization conditions:

1. *BufferCnt ≥ 0*
2. *BufferCnt ≤ BufferSize*

By **topology invariant**, we know that *BufferCnt = ?*

Example of a synchronization problem


```
mutex m := 1;
int BufferCnt := 0;
```

```
Task Producer =
|[ while( true ){
        produce_items();

        lock(m);
   G    BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
}
]|
```

||

```
Task Consumer =
|[ while( true ){

        lock(m);
   C    BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        use_items();
  }
]|
```

Equivalent synchronization conditions:

1. *BufferCnt* ≥ *0*
2. *BufferCnt* ≤ *BufferSize*

By **topology invariant**, we know that
$BufferCnt = 0 + 4 \times cG \; - 2 \times cC$

Example of a synchronization problem


```
mutex m := 1;
int BufferCnt := 0;
```

```
Task Producer =
|[ while( true ){
        produce_items();

        lock(m);
    G   BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
}
]|
```

||

```
Task Consumer =
|[ while( true ){

        lock(m);
    C   BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        use_items();
  }
]|
```

Equivalent synchronization conditions (injecting the topology invariant):

1. $4 \times cG - 2 \times cC \geq 0$
2. $4 \times cG - 2 \times cC \leq \mathit{BufferSize}$

Example of a synchronization problem


```
mutex m := 1;
int BufferCnt := 0;
```

```
Task Producer =
|[ while( true ){
        produce_items();

  G {   lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
}
]|
```

||

```
Task Consumer =
|[ while( true ){

  C {   lock(m);
        BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        use_items();
  }
]|
```

Equivalent synchronization conditions

1. $2 \times cC \leq 4 \times cG$
2. $4 \times cG \leq 2 \times cC + BufferSize$

Given: a *synchronization condition:* $\sum_i a_i \times cA_i \leq \sum_j b_j \times cB_j + e$ for non-negative constants a_i, b_j, and e .

Solution:

- introduce **semaphore** ***s***, with **initial value** $s_0 = e$
- **replace action** A_i **with** $P(s)^{a_i}\, A_i$
- **replace action** B_j **with** $B_j\, V(s)^{b_j}$

Example of a synchronization problem


```
mutex m := 1;
int BufferCnt := 0;
```

```
semaphore s1 := 0;
semaphore s2 := BufferSize;
```

Introduce one semaphore per synchronization condition

```
Task Producer =
|[ while( true ){
        produce_items();
        P(s2)^4;
    G { lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

||

```
Task Consumer =
|[ while( true ){
        P(s1)^2;
    C { lock(m);
        BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        V(s2)^2;
        use_items();
}
]|
```

Equivalent synchronization conditions

1. $2 \times cC \leq 4 \times cG$
2. $4 \times cG \leq 2 \times cC + BufferSize$

Given: a *synchronization condition:* $\sum_i a_i \times cA_i \leq \sum_j b_j \times cB_j + e$
for non-negative constants a_i, b_j, and e .
Solution:
- introduce **semaphore *s*,** with **initial value $s_0 = e$**
- **replace action A_i with $P(s)^{a_i}\ A_i$**
- **replace action B_j with $B_j\ V(s)^{b_j}$**

Exercise
(typical exam question)

Given: a *synchronization condition:* $\sum_i a_i \times cA_i \leq \sum_j b_j \times cB_j + e$
for non-negative constants a_i, b_j, and e .
Solution:
- introduce **semaphore *s*,** with **initial value $s_0 = e$**
- **replace action A_i with $P(s)^{a_i}\ A_i$**
- **replace action B_j with $B_j\ V(s)^{b_j}$**

```
int f := 5;
Int w := 0;
```

```
Task T =
|[ while( true ){
        transport();
        f := f+1;
        w := w+3;
}
]|
```

||

```
Task P =
|[ while( true ){
        w := w-2;
        produce();
        f := f+1;
}
]|
```

||

```
Task S =
|[ while( true ){
        f := f-2;
        transport();
        w := w+3;
}
]|
```

Question:
Enforce the following synchronization invariants:
- **I1:** $w \geq 0$
- **I2:** $f \geq 2 \times w$

Prevent race conditions on *f* and *w*
Functions *produce()* and *transport()* cannot be in critical sections

Exercise (typical exam question)

```
int f := 5;
Int w := 0;
```

```
Task T =
|[ while( true ){
       transport();
       T1: f := f+1;
       T2: w := w+3;
}
]|
```

||

```
Task P =
|[ while( true ){
       P1: w := w-2;
       produce();
       P2: f := f+1;
}
]|
```

||

```
Task S =
|[ while( true ){
       S1: f := f-2;
       transport();
       S2: w := w+3;
}
]|
```

f = cT1 +cP2 – 2*cS1 + 5
w = 3*cT2 – 2*cP1 + 3*cS2 + 0
Thus, the invariants require
I1 : 2*cP1 <= 3*cT2 + 3*cS2
I2 : 6*cT2 + 2*cS1 + 6*cS2 <= cT1 +cP2 + 4*cP1 + 5

We need 2 semaphores s1, s2 and two mutexes mf and mw initialized as follows:
s1:=0; s2:=5; mf:=1; mw:=1;
We then replace the actions as follows:
T1 → P(mf) T1 V(mf) V(s2)
T2 → P^6(s2) P(mw) T2 V(mw) V^3(s1)
P1 → P^2(s1) P(mw) P1 V(mw) V^4(s2)
P2 → P(mf) P2 V(mf) V(s2)
S1 → P^2(s2) P(mf) S1 V(mf)
S2 → P^6(s2) P(mw) S2 V(mw) V^3(s1)

Exercise (typical exam question)

```
int f := 5;
Int w := 0;
```

```
semaphore s1 := 0;
semaphore s2 := 5;
```

```
mutex mf := 1;
mutex mw := 1;
```

```
Task T =
|[ while( true ){
        transport();

        P(mf);
        f := f+1;
        V(mf);
        V(s2);

        P(s2)^6;
        P(mw);
        w := w+3;
        V(mw);
        V(s1)^3;
   }
]|
```

||

```
Task P =
|[ while( true ){
        P(s1)^2;
        P(mw);
        w := w-2;
        V(mw);
        V(s2)^4;

        produce();

        P(mf);
        f := f+1;
        V(mf);
        V(s2);
   }
]|
```

||

```
Task S =
|[ while( true ){
        P(s2)^2;
        P(mf);
        f := f-2;
        V(mf);

        transport();

        P(s2)^6;
        P(mw);
        w := w+3;
        V(mw);
        V(s1)^3;
   }
]|
```

Correctness of the proposed solution

- Since we have solved a synchronization problem and introduced blocking we must verify the correctness criteria.

- **Functional correctness** and
  minimal waiting are by construction.

- **Fairness**: the solution is just as fair as the semaphores and mutex.
  - depends on semaphore implementation and order of release from the waiting queue.

- **Absence of deadlock**: see next section

Agenda

- Proving program properties using invariants
- Actions synchronization using semaphores
- Preventing deadlocks

Reasoning about deadlock

- **A deadlock state** is a system state in which ***a set of tasks*** is **blocked *indefinitely***
  - **each task is blocked on another task in the same set**
- We typically **prove** the absence of deadlock **by contradiction**
  - assume a deadlock occurs
  - investigate all task sets that can be blocked at the same time (often: just 1)
  - show a contradiction for all possible combinations of blocking actions of those tasks

Proof example

```
mutex m := 1;
int BufferCnt := 0;
```

```
semaphore s1 := 0;
semaphore s2 := BufferSize;
```

```
Task Producer =
|[ while( true ){
        produce_items();
        P(s2)^4
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

Assume we are blocked here indefinitely …

… then **another task must hold the mutex** forever.

It can only happen here.

However, it is **impossible for *Consumer* to be blocked forever in its critical section** (there is no blocking operation)**,** and ***Consumer* always calls unlock(m) at the end the critical section** (i.e., releases the mutex)

```
Task Consumer =
|[ while( true ){
        P(s1)^2;
        lock(m);
        BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        V(s2)^2;
        use_items();
}
]|
```

A similar argument must be built for any other **point were Producer or Consumer may be blocked**

➔ **Not a deadlock**

When can it go wrong?

Let's swap *P(s1)* and *lock(m) in Consumer*

mutex m := 1;
***int** BufferCnt := 0;*

semaphore s1 := 0;
semaphore s2 := BufferSize;

```
Task Producer =
|[ while( true ){
        produce_items();
        P(s2)^4
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

```
Task Consumer =
|[ while( true ){
        lock(m);
        P(s1)^2;
        BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        V(s2)^2;
        use_items();
}
]|
```

What can go wrong?

Let's swap *P(s2)* and *lock(m) in Consumer*

```
mutex m := 1;
int BufferCnt := 0;
```

```
semaphore s1 := 0;
semaphore s2 := BufferSize;
```

```
Task Producer =
|[ while( true ){
        produce_items();
        P(s2)^4
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

Assume we are blocked here indefinitely …

… then **another task must hold the mutex** forever.

It can only happen here.

***Consumer* could be blocked here** (i.e., buffer is not full enough)

Since Producer is blocked, it ***cannot call V(s1).*** *Therefore,* ***Consumer*** will remain blocked and will be **unable to call unlock(m)**

```
Task Consumer =
|[ while( true ){
        lock(m);
        P(s1)^2;
        BufferCnt := BufferCnt-2;
        pop_items();
        unlock(m);
        V(s2)^2;
        use_items();
}
]|
```

➔ **Possible deadlock**

Note: for a full proof that a deadlock can be reached, a trace should be produced

Preventing deadlock

- Always let critical sections terminate
  - always **call *unlock(m)* after *lock(m)***
- **Avoid cyclic waiting**
  - Try to avoid *P or lock* operations that may block indefinitely between *lock(m) and unlock(m)*

Cyclic waiting

```
mutex m1 := 1;
mutex m2 := 1;
mutex m3 := 1;
```

```
Task T1 =
|[ while( true ){
        P(m1);
        P(m2);
        doSomething();
        V(m2);
        V(m1);
}
]|
```

```
Task T2 =
|[ while( true ){
        P(m2);
        P(m3);
        doSomething();
        V(m3);
        V(m2);
}
]|
```

```
Task T3 =
|[ while( true ){
        P(m3);
        P(m1);
        doSomething();
        V(m1);
        V(m3);
}
]|
```

What can go wrong?

Cyclic waiting

```
mutex m1 := 1;
mutex m2 := 1;
mutex m3 := 1;
```

```
Task T1 =
|[ while( true ){
        P(m1);
        P(m2);
        doSomething();
        V(m2);
        V(m1);
}
]|
```

```
Task T2 =
|[ while( true ){
        P(m2);
        P(m3);
        doSomething();
        V(m3);
        V(m2);
}
]|
```

```
Task T3 =
|[ while( true ){
        P(m3);
        P(m1);
        doSomething();
        V(m1);
        V(m3);
}
]|
```

(**T1.P(m1)**) {m1=0} (**T2.P(m2)**) {m2=0} (**T3.P(m3)**) {m3=0}

T1 **blocked** on *P(m2)*, *T2* **blocked** on *P(m3)*,
T3 **blocked** on *P(m1)*
➔ **deadlock**

Cyclic waiting
solution: lock semaphores in a fixed order

```
mutex m1 := 1;
mutex m2 := 1;
mutex m3 := 1;
```

Swap *P(m1)* and *P(m3)*

```
Task T1 =
|[ while( true ){
        P(m1);
        P(m2);
        doSomething();
        V(m2);
        V(m1);
}
]|
```

```
Task T2 =
|[ while( true ){
        P(m2);
        P(m3);
        doSomething();
        V(m3);
        V(m2);
}
]|
```

```
Task T3 =
|[ while( true ){
        P(m1);
        P(m3);
        doSomething();
        V(m1);
        V(m3);
}
]|
```

Solution: always call *P(m3)* after *P(m2)* which is always called after *P(m1)*

Exercise: prove the absence of deadlock for this program (**exam 2020-2021**)

Preventing deadlock

- Always let critical sections terminate
    - always **call *unlock(m)* after *lock(m)***
- **Avoid cyclic waiting**
    - Try to avoid *P or lock* operations that may block indefinitely between *lock(m) and unlock(m)*
    - Use a **fixed order when calling *P*-operations on semaphores or mutexes**
        - *P(m);P(n); ....* in one task may deadlock with *P(n);P(m);...* in another task
    - **Look for the "*dining philosopher problem*" for a good example**

Assume multiple *Producers*

Assume
BufferSize = 4

mutex m := 1;
***int** BufferCnt := 0;*

semaphore s1 := 0;
semaphore s2 := 4;

```
Task Producer1 =
|[ while( true ){
       produce_items();
       P(s2)^4
       lock(m);
       BufferCnt := BufferCnt+4;
       push_items();
       unlock(m);
       V(s1)^4;
}
]|
```

```
Task Producer2 =
|[ while( true ){
       produce_items();
       P(s2)^4
       lock(m);
       BufferCnt := BufferCnt+4;
       push_items();
       unlock(m);
       V(s1)^4;
}
]|
```

What can go wrong?

Assume multiple *Producers*

Assume
BufferSize = 4

mutex m := 1;
int *BufferCnt := 0;*

semaphore s1 := 0;
semaphore s2 := 4;

```
Task Producer1 =
|[ while( true ){
        produce_items();
        P(s2)^4
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

Assume ***Producer1*** **executes** ***P(s2)*** **two times**…

… then *s2 = 2*

Now, assume ***Producer2*** **executes** ***P(s2)*** **two times**

… then *s2 = 0*

Both tasks are blocked

```
Task Producer2 =
|[ while( true ){
        produce_items();
        P(s2)^4
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

➔ **deadlock**

Preventing deadlock

- Always let critical sections terminate
  - always **call *unlock(m)* after *lock(m)***
- **Avoid cyclic waiting**
  - no *P or lock* operations that may block indefinitely between *lock(m) and unlock(m)*
  - Use a **fixed order when calling *P*-operations on semaphores or mutexes**
    - *P(m);P(n); ....* in one task may deadlock with *P(n);P(m);...* in another task
  - Look for the "*dining philosopher problem"* for a good example
- **Avoid greedy consumers**
  - ***P(a)***^{k} should be **an indivisible atomic operation** when tasks compete for limited resources

Assume multiple *Producers*: solution

```
mutex m := 1;
int BufferCnt := 0;
```

```
semaphore s1 := 0;
semaphore s2 := 4;
mutex mp := 1;
```

New mutex

```
Task Producer1 =
|[ while( true ){
        produce_items();
        lock(mp); P(s2)^4; unlock(mp);
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

||

```
Task Producer2 =
|[ while( true ){
        produce_items();
        lock(mp); P(s2)^4; unlock(mp);
        lock(m);
        BufferCnt := BufferCnt+4;
        push_items();
        unlock(m);
        V(s1)^4;
}
]|
```

Summary

- **Proving program properties** using **topology invariants**
- **Synchronize tasks** to enforce new properties **using semaphores**
- How to **prove the existence or absence of deadlock**
- **Rules of thumb to prevent deadlocks** when using semaphores and mutexes

- **We come back to deadlocks in two weeks**

- **Additional exercises will be available on Canvas to train yourself at actions synchronization and generating traces**
  - **Work on them! They will prepare you for the exam.**

- **Homework of next week will be on invariants and action synchronization**